JAVA Agents for Distributed System Management

Christopher Brooks

University of Michigan

Brian Tierney and William Johnston

Lawrence Berkeley National Laboratory[1]
University of California, Berkeley, CA, 94720

Abstract

Wide-area distributed systems provide significant benefits in terms of parallelism, robustness and accessibility. However, they also present a new challenges in terms of configuration, maintenance and monitoring. A semi-autonomous management mechanism is needed to deal with systems that have lost contact with central administrators or have to deal with problems on time scales not possible by human administrators. In this paper we describe the use of Java-based agents and brokers to control and monitor a complex, highly distributed parallel application These agents use high-level reasoning to solve problems and perform automated tasks. They also use IP-multicast to send KQML messages to other associated agents - an “agency”. We describe an architecture and design, and discuss the advantages and disadvantages of using Java for this purpose. We also show how these agents are extremely useful for performance monitoring of distributed systems.

Introduction

Current research at Lawrence Berkeley National Laboratory is exploring the use of highly distributed computing and storage architectures to provide all aspects of collecting, storing, analyzing, and accessing large data-objects. These data-objects can be anywhere from tens of MBytes to tens of GBytes in size. They are typically the result of a single operational cycle of an instrument, such as single large images from electron microscopes, video images from cardio-angiography, sets of related images from MRI procedures, and images and numerical data from a particle accelerator experiment. The sources of such data objects, e.g., centralized health care facilities or large scientific instruments, are often remote both from the users of the data and from available large-scale storage and computation systems.

Our Wide-Area network-based Large Data-Object (WALDO) architecture [3] utilizes a high-speed wide-area network between the object sources and a multi-level distributed storage system. As the data is being stored, a cataloguing system automatically creates and stores condensed versions of the data, textual metadata and pointers to the original data. The catalogue system provides a Web-based graphical interface to the data.

1. The work described in this paper is supported by ARPA, Computer Systems Technology Office (http://ftp.arpa.mil/ResearchAreas.html) and the U. S. Dept. of Energy, Office of Energy Research, Office of Computational and Technology Research, Mathematical, Information, and Computational Sciences Division (http://www.er.doe.gov/production/octr/mics), under contract DE-AC03-76SF00098 with the University of California. Authors: chbrooks@eecs.umich.edu, wejohnston@lbl.gov, bltierney@lbl.gov, Lawrence Berkeley National Laboratory, mail stop: B50B-2239, Berkeley, CA, 94720, http://www-itg.lbl.gov). This is report no. LBNL-NNNNN.

Motivation

Applications operating in a distributed environment within a wide-area network often need knowledge about the state of the network and the hosts within it in order to operate efficiently. However, much of this knowledge, such as the latency between two remote hosts or the number of users on a particular server, is not easily determined from a single node within the network. Furthermore, when things go wrong with the network or the servers, one frequently cannot access the state of the system after the fact -- some sort of continuous state monitoring is needed to maintain a global history of the system. Additionally, it is desirable for this monitoring system to be generalized and adaptable: new types of users or applications may have different monitoring requirements. Analysis of the events leading to congestion or system failure may also require accessing different types of data at different granularities than those examined during normal operation, thus requiring adaptability of the monitors. The system must also have a great deal of autonomy. It must be able to make decisions and execute tasks without prompting a human user.

One solution to these problems is to use a collection of software agents [2] to provide structured access to current and historical information. In this paper we define an **agent** to be a process that monitors the state of the system. These agents only communicate with other agents, not with clients. We define a **broker agent** (or **broker**) to be an agent that manages this information, filters information for clients, or performs some action on behalf of a client.

These agents are autonomous, adaptable entities that are capable of filtering information about the state and history of a system, such as the throughput between servers, the number of users in a system, or the location of different copies of a given dataset. They are also able to maintain a continually updated view of the global state of the system, which allows users to optimize the configuration used depending on their particular requirements. In addition, they are able to independently perform sets of administrative tasks, such as the restarting of server processes.

This agent system, known as WHERE, is written entirely in Java, with the exception of a small native library written in C. Java provides several features that make assist in the development and operation of a distributed agent environment. Java's clean, object-oriented style lends itself nicely to the high-level AI programming style used in WHERE. Additionally, the support for dynamic loading of classes provided by Java's reflection package is essential to WHERE's ability to learn and propagate new behaviors. Also, Java's support for TCP and multicast allows WHERE agents to communicate efficiently with each other and third-party applications across a wide-area network.

The Distributed Parallel Storage System

An important aspect of the WALDO architecture is the ability to access very large data-objects quickly. To this end, we have designed and implemented a disk-based caching system called the Distributed-Parallel Storage System (DPSS) [5]. The DPSS is a collection of wide area distributed disk servers that operate in parallel to provide high-speed logical block level access to large data sets. These data sets are broken up into 64 KB blocks that are declustered (dispersed in such a way that as many system elements as possible can operate simultaneously to satisfy a given request) across both disks and servers. This strategy allows both a large collection of disks to seek in parallel, and all servers to operate in parallel to send the requested data to the application, enabling the DPSS to perform as a high-speed data block server.

The DPSS consists of a name server, where the logical block names are translated to physical addresses (server: disk: disk offset), and several disk servers. In the disk servers, the data is read from disk into local cache, and then sent to the applications.

The DPSS was developed as part of the DARPA funded MAGIC Testbed project (see: http://www.magic.net). We have maintained DPSS configurations spanning separate geographic and administrative domains. These experiences have demonstrated the difficulties in configuring and maintaining widely distributed systems by hand.

Agent Usage in a DPSS

A distributed system such as the DPSS, with several components, can greatly benefit from the use of monitoring agents that provide structured access to current and historical information regarding the state of the DPSS. Agents can provide DPSS clients with different views and monitoring granularities of the system's current and past state, depending on a client's needs. For example, DPSS clients that need reliable high-speed connections for a long period of time, such as clients that capture the results of a particle accelerator experiment, would use an agent that was capable of collecting and organizing detailed network statistics. Also, if an agent detects some sort of failure or congestion, it is able to automatically change the granularity at which it collects data, providing the DPSS administrator with a more detailed picture of the system at the time the anomaly or failure occurred.

Agents can keep track of all components within the system and restart any component that has gone down, including any DPSS server or one of the other agents. This greatly improves the system's fault tolerance.

The agents also monitor the load of each DPSS block server. The DPSS allows for replication of data on multiple servers. If the load on one server is unacceptably high, the agents can be used to decide which alternate server to use. Each client provides a set of constraints that the agents use to recommend a server configuration to a client. Some clients may be interested in network latency, while others may be concerned about the total time required to store a set (which is dependent on the number of servers used and the load on each server). As these parameters change, the agents are able to track them and determine whether other configurations would be better for a particular client.

Brokers can be used to keep track of multiple DPSS's. Clients may wish to replicate their data across multiple DPSS's and choose between them depending on the location of the client at a given time. Brokers know which data sets are loaded on each DPSS and which DPSS has the best network connectivity to a given client, and can advise the client which is the best DPSS to use.

A broker/agent architecture allows the system administrators to separate mechanism from policy. Agents and brokers can be used to determine policies and rules to be enforced while remaining completely separate from the actual mechanism used to implement these policies. Agents and brokers are able to exchange information at a high level, hiding the details of how a particular policy, such as security, is implemented. For example, agents can be used to provide secure authorization for DPSS clients. The client only needs to be aware of the security policy, which consists of obtaining a security certificate from a known server and presenting it to the DPSS's broker, without needing to know anything about the mechanism by which public keys are generated or how a certificate is actually generated. One an acceptable policy is in place, administrators are free to alter the underlying mechanisms without affecting clients.

When new components, such as a new disk server, are added to a DPSS configuration, the agents do not have to be reconfigured or restarted. When an agent is started on the new host it will inform all other agents about the new server. Agents are able to continually propagate information about the state of the system to each other, such as the addition and deletion of hosts, disks and interfaces.

New agent methods can be added at any time. Agents are capable of informing each other about new tasks to be performed or about changes in existing tasks. For example, the brokers have an algorithm

for determining which DPSS configuration to use based on a set of parameters that include network bandwidth, latency and current server load. If desired, this algorithm may be modified “on the fly” without having to reinstall a new set of binaries on every host in the system. A DPSS administrator can also design a new task that automatically moved all datasets on a server the were more than one week old to tertiary storage. Once this was “taught” to one agent, that agent will then propagate the task to all the other agents in the agency, saving the administrator from needing to manually reconfigure each agent.

An agent can also be used to assist in the management of data sets. For example, an agent can be instructed to replicate data sets across a second set of disks, automatically compress image sets, migrate sets to mass storage after a given time or keep track of the content of an evolving dataset. The organization of agents to accomplish this sort of thing is shown in Figure 1.

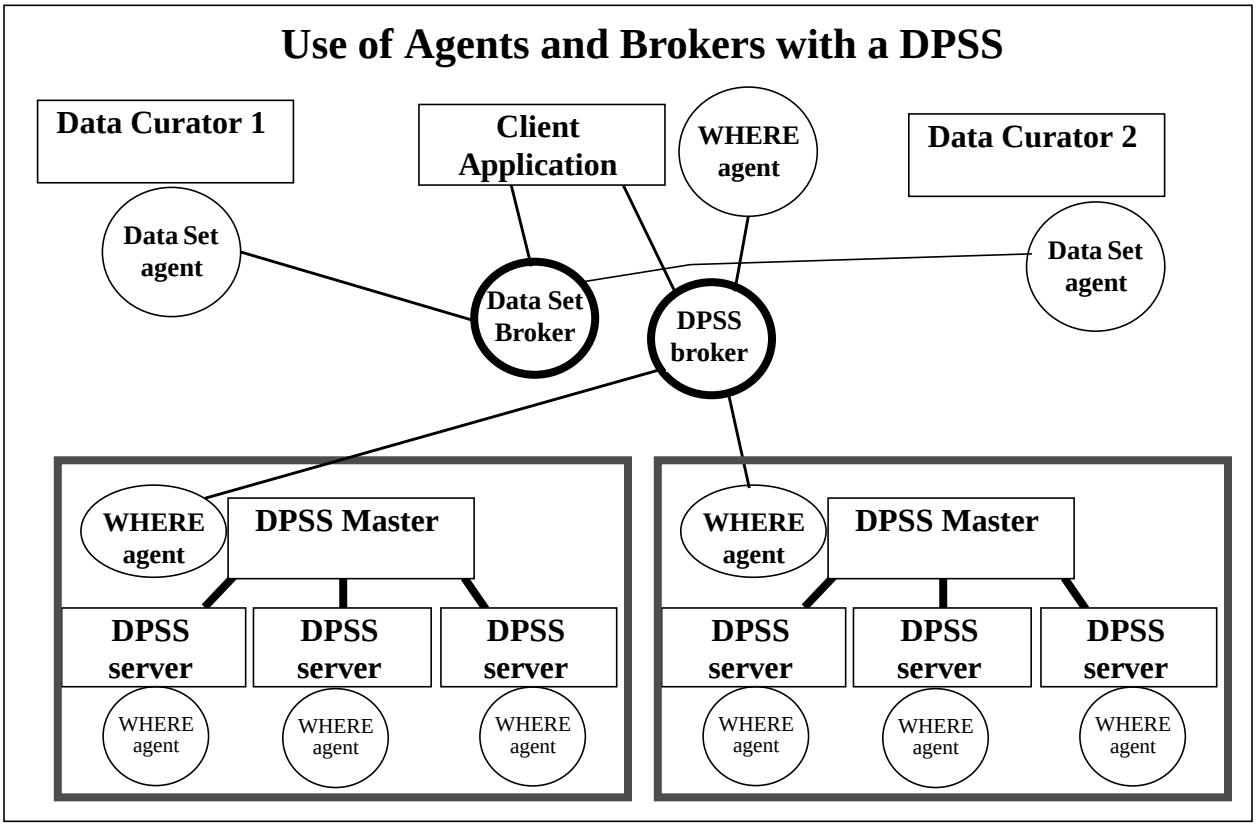

Figure 1 DPSS use of agents and brokers

WHERE Architecture

WHERE is a distributed system of agents written in Java. These agents model their environment using an extensible set of **Facts**. These Facts are then used to help the agents make decisions. WHERE agents act on their environment using a set of **Tasks**. These Tasks may either accomplish a goal directly or be strung together in a sequence called a plan in order to accomplish more complicated goals.

A WHERE agency consists of a set of agents who are overseen and coordinated by a broker. WHERE agents start out as generalized programs with a limited set of capabilities. One of these capabilities is the ability to dynamically load and execute new code. They are then trained with a particular set of

tasks to allow them to perform a specific function, such as monitoring a host's network traffic or restarting a server that has crashed.

WHERE brokers are provided with additional tasks that allow them to filter and collate data for clients. Brokers serve as the agency's interface with a client or another agency. Generally, brokers are responsible for filtering, collecting, managing and maintaining information gathered by agents. They distribute client requests for information to the appropriate agents and filter and collect the results for return to the client. Brokers are also responsible for assembling and maintaining a consistent view of the system from a number of independent sources of information.

Agent Model

WHERE agents reason generally about a closed-world environment composed of servers, data sets, routes, hosts and clients. They collect Facts about their world and use them to attempt to construct a consistent picture of the state of a DPSS and determine what actions are needed in order to achieve their goals. These Facts include the throughput between two nodes in a network, the number of users on a host, the datasets stored on a particular server, and the IP address of a particular client, among other things. This collection of types of knowledge that a WHERE agent can reason about is referred to as its **ontology.** The fact that WHERE agents have a limited, well-defined ontology allows them to use general reasoning techniques with a good chance of success.

Additionally, agents (and brokers) maintain two other types of knowledge: Tasks and Operators.

A Task is an action that an agent is able to perform. The only requirement is that it implement the method evaluate, which the agent calls to execute the task. Currently implemented tasks include: findPathTask, which finds the fastest route between two hosts, getServerInfoTask, discovers information about a DPSS, getHostInfoTask, which gets information about a host and checkServerTask, which probes a DPSS server to see if it is running and restarts it if it isn't. Tasks are used to enable agents to perform simple, routine maintenance that needs to be done regularly and repeatedly.

WHERE agents satisfy more complicated goals by using **Operators** to construct a **Plan**. This is accomplished by using a partial-order planner[6] to implement a problem-solving strategy known as goal-based reasoning. In goal-based reasoning, the agent constructs a state representing the current state of the world from the Facts that it knows. It is asked to accomplish some goal, which is also modeled as a state that is composed of Facts about its world. These states may be viewed as nodes in a graph, while available Operators may be viewed as edges. The problem then becomes one of finding a path from one node in a graph to another, where every node represents a possible state.

Since there are too many possible states to instantiate them all and find a path directly, a different approach must be used. A partial-order planner is used to find a path from the start state to the goal state, if one exists. Since the goal state contains one or more conditions that must be met, the planner searches through the known Operators to find one that has a postcondition that matches one of the conditions of the goal state. (If there is no such Operator, then the goal cannot be reached.) This Operator may have its own preconditions that must be satisfied, and so the planner searches for an Operator that will either satisfy one of these conditions or one of the goal conditions. This continues until all preconditions have been satisfied. The Operators chosen are placed into a partial ordering that preserves the preconditions needed for each operator. The agent then executes each of these Operators in turn to reach the goal state, thereby accomplishing what it was asked to do.

This knowledge-based approach [4] provides two benefits: First, it allows WHERE agents to be extended to perform new tasks and thus prevents 'brittleness'. Rather than only being able to perform a few select tasks, the agent is able to attempt to solve any problem that can be expressed as a goal

state. The agents' only limitation are the Operators that are available to it. Providing the agent with new Operators allows it to perform new tasks. It also allows Operators to be reused for solving similar tasks. Many of the diagnostic and monitoring tools used for the DPSS have a great deal of common code. Expressing this code as a set of Operators allows it to be reused within different contexts.

Second, this approach allows the central agent codebase to remain small. New agent capabilities ("**behaviors**") can be added by providing the agent with new operators and Tasks. These new operators increase the number of possible goal states the agent is able to achieve, thereby increasing its functionality. New behaviors can be loaded dynamically and propagated between agents, allowing the agency as a whole to increase its store of knowledge while it is running. This ability to learn to perform new types of tasks allows users to adapt and extend WHERE for their own particular needs.

Communication

WHERE agents and brokers use a communication language called KQML [1]. KQML provides an extensible set of high-level agent messages known as performatives. These performatives allow agents and brokers to communicate information and meta-information in a format that is completely separate from the agent's internal representation of this information. This makes it possible for WHERE agents and brokers to interact with other agents that may have an entirely different internal representation of their world.

KQML messages are structured using a Lisp-like syntax. Each message begins with a performative, followed by a property list indicating the sender and receiver of the message, how to interpret it, and the content of the message.

A performative is a request for an agent to perform a specific task, such as evaluating a function, adding a Fact to its knowledge base or answering a query. Common performatives include ask-about, insert, evaluate and reply.

A sample KQML message used in WHERE is shown below:

```
(ask-about: sender Broker: receiver lbl-server3
     :language KQML: ontology DPSS: reply-with reply-tag
     :content (type host lbl-server3))
```

This is a request from a WHERE broker to an agent asking for any information it has about the host it is running on. The performative ask-about asks the receiver to return a Fact to the sender. The remainder of the message is a property list, that can be broken into tag-value pairs.

KQML is based on a connectionless, best-effort delivery system. An agent sends a message and hopes for a reply. The receiver is under no obligation to send a reply, or to even acknowledge that the message was received. This design spares agents from having to maintain lists of pending communications or state tables tracking their communication with other agents. A reply to a communication can be considered to be an acknowledgment, and a lack of reply can be considered to be a negative reply. (i.e. the agent is unable to satisfy the performative.)

It is worth noting that the KQML design is independent of the underlying network protocol used to deliver these messages. An implementation could use TCP to ensure delivery of all messages; this has no bearing on an agent's requirement to respond to a KQML message.

Inter-agent communication in WHERE is provided by IP multicast; all agents within an agency are part of the same multicast group. This provides a logical bus for intra-agency communication. Messages are multicast on this bus, allowing one agent to refer to another only by its name (that is typi-

cally associated with the canonical name of the host it runs on), rather than needing to explicitly know the DNS name of the host the other agent is running on. All agents will receive a copy of every message; messages received by an agent that are not intended for it are discarded.

Multicast also allows agents to easily disseminate new Tasks and Operators. The agent simply sends an add-task performative on the bus, indicating that all agents should receive it. The agent only needs to send one copy of the message; the underlying multicast routers will copy it and forward it to all other agents in the multicast group.

Use of Java within WHERE

Java provides five capabilities that are essential to WHERE's functionality. They are: portability, object-oriented design, reflection and dynamic loading, support for both unicast (TCP) and multicast, and the ability to interface with existing C libraries.

Portability is essential to WHERE's success. DPSS's are heterogeneous systems, and maintaining separate code trees for every possible architecture is a difficult and time-consuming task. Additionally, the portability of Java's data types makes it simple for agents on one platform to exchange data with agents on another platform without being concerned about byte ordering, integer representation, or any of the other issues that typically plague developers of cross-platform communications. Portability also makes it possible for agents to exchange code with each other, a central feature of WHERE.

Java's clean object-oriented design is also essential to the structure of WHERE. WHERE's structure is based on a set of hierarchically-related agents that exchange various Facts about the world. Treating these Facts as objects allows agents to use them to compactly represent related types of information while treating them homogeneously for the purposes of storage, retrieval and exchange. An object-oriented approach also makes it easy to add new Tasks and Facts to an agent's ontology. Since the Tasks and Facts are subclasses of an existing abstract class, the addition is transparent to the agent, which just treats them as an instance of the parent class.

Reflection and dynamic loading are key characteristics that allow WHERE agents to acquire new behaviors and take on new tasks. This is quite probably Java's greatest strength with regard to WHERE. Since WHERE agents may be widely distributed across the Internet, is not feasible to log in to the machine on which they are running, recompile the source code and restart the agent. Dynamic loading allows the agent to load new Tasks as they are needed, keeping the code base small and allowing for incremental updating. Reflection allows WHERE agents to send code from one agent to another; the receiving agent is able to query the new class to discover its name, methods and the arguments of those methods. *This allows a WHERE designer to add a new class to an agency by informing a single agent of its location*; this agent will then inform all the other agents it knows about, who can then upload the code from a given URL.

Since the DPSS has a large body of existing C libraries for communication and data manipulation, a successful agent implementation should take advantage of this whenever possible. WHERE does this using Java's native methods. This allows WHERE agents to communicate with the DPSS using existing libraries, solving a potentially thorny problem: the DPSS was originally written to use XDR as the protocol for exchanging data. However, there is no stable version of an XDR converter in Java. Rather than write one from scratch, we were able to create a native method wrapper for these routines, allow-

ing WHERE to communicate seamlessly with the DPSS. The one issue here is portability: since this code is written in C, it will only run on architectures for which it has been compiled. However, dynamic loading softens this blow. Since Java methods are loaded at runtime, only those agents that will actually need this method (a fairly small percentage) will suffer from this restriction.

Java is not perfect. There are several problems with the language and environments that can make it difficult to use effectively. Two problems that we consider are lack of debugging support and efficiency.

Debugging support for multi-threaded Java programs in the Unix environment is nearly non-existent. The Java debugger, *jdb* is simply unable to help when working with multi-threaded programs. It cannot switch between thread contexts or set breakpoints accurately. We had some success using Sun's Java Workshop to debug Java programs (although it is rather slow and buggy) but at the time of development, we were using Java v1.1 for WHERE, and Java Workshop only supported Java v1.0.2. As a result, most debugging was done using print statements and auxiliary programs to probe agents for their state. This was often a slow and painful process.

Java also has some well-known problems with efficiency. This is to be expected; since it is an interpreted language, it cannot compete with native code applications, nor should it be expected to. In most cases, we were able to avoid these problems by reducing the amount of computation done by WHERE agents. However, there were cases where Java's interface with the operating system caused problems. One example is Java's FileInputStream class. It has a method called skip() that reads over and skips n bytes in a File. While it seems that this would provide functionality similar to that of Unix's lseek(), in fact, it merely reads the characters and discards them, rather than incrementing the file pointer. This makes a huge difference to an agent trying to find the last line of a 10M file!

Results

WHERE is currently being used for three purposes: to provide current network status information to client applications, to monitor and restart DPSS servers, and to collect and display near-real-time statistics about a DPSS.

In a research environment, DPSS servers are have multiple networks interfaces (i.e.: ethernet, FDDI, and ATM), one or more of which may not be up at any given moment. The agents are used to test all possible network interfaces every few minutes, and keep track of the fastest available interface on each server at any given moment. When a DPSS client requests a data set from a DPSS, it queries the broker associated with that DPSS's agency and find the fastest interface (Ethernet, ATM, etc.) between the client and server machines. Without the agents, the clients would need to try all possible interface to determine the fastest, a process that can take several seconds, or even longer in a wide-area ATM environment where call setup time may take up to one minute per connection.

WHERE agents are also able to continuously monitor DPSS servers. When an agent detects that a server has failed, it is able to attempt to restart it. If it is unable to restart the server, it can take an alternate action, such as sending email to the administrator of this DPSS informing them of the problem. This autonomy makes the DPSS much more robust and eases the duties of a DPSS administrator.

WHERE agents and brokers are also able to provide a near-real-time picture of the DPSS, its clients, and the relative usage of various servers and network connections. The broker collects these statistics from agents throughout the system, collates and massages them and forwards them to an applet that provides users with a graphical representation of the system, as shown in Figure 2. Previously these statistics were available, but difficult to access in real time. A user could determine after the fact how the system had changed by using the logs and graphs generated from them, but had little means of watching the effects of an experiment on the system while the experiment was taking place. WHERE provides DPSS users with that capability.

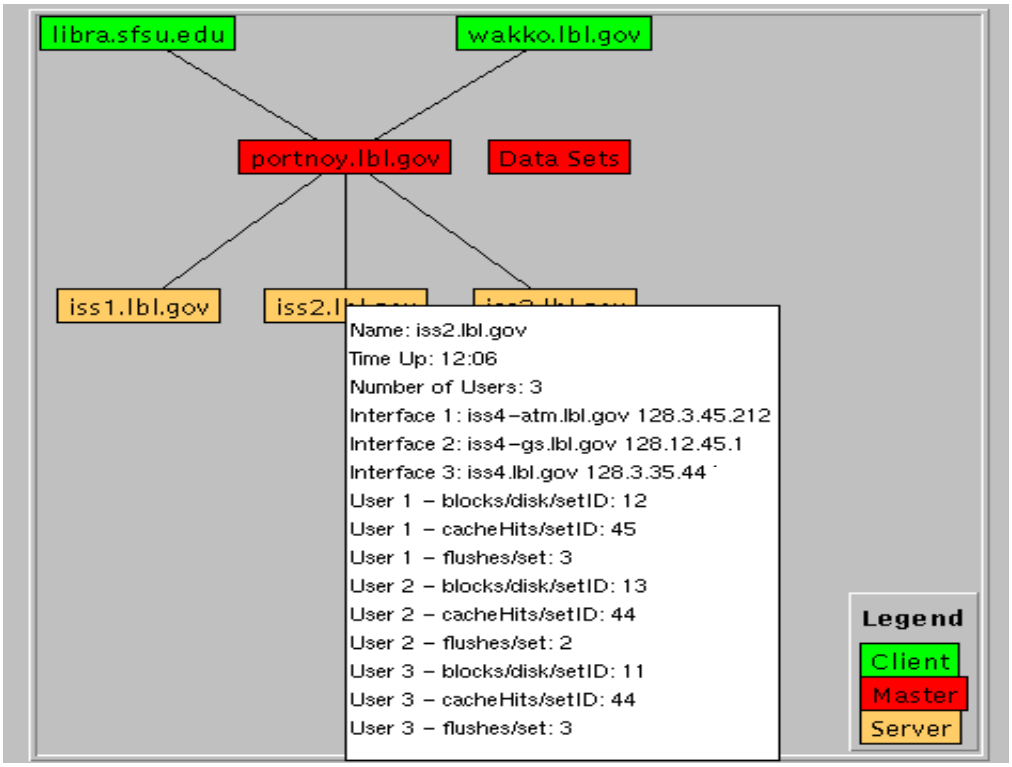

Figure 2 Sample output from system status applet

In the near future we plan to add new tasks to keep track of where various data sets are located (i.e.: which DPSS or which tertiary storage location), and write a broker to automatically load or unload sets from the DPSS. We also will write tasks to manage log files generated by various distributed system components, and provide a mechanism to query the agents to look for certain types of log entries.

Conclusion

We have found that agents have proved to be quite useful in monitoring and maintaining highly distributed systems. They allow users and administrators to maintain a global view of the system, even if the components are geographically and administratively separate. WHERE's high-level knowledge-based design, which uses Facts, Tasks and Operators to determine a course of action, rather than hardcoding actions into the agents, allows the agents to grow and adapt to a user's needs. Java plays an integral role in this architecture, allowing us to design a highly adaptable, portable set of software agents with high-level behaviors and mature networking capabilities that are also capable of interfacing with existing code written in other languages.

For More Information

More information is available on the WWW at:
http://www-itg.lbl.gov/DPSS/agents/WHERE.html
http://www-itg.lbl.gov/DPSS
http://www-itg.lbl.gov/WALDO

References

[1] Finin, T. et. al., DRAFT Specification of the KQML Agent Communication Language, unpublished draft, 1993. (http://www.cs.umbc.edu/kqml/kqmlspec/spec.html)

[2] Genersereth, Michael and Ketchpel, Steven, Software Agents, Communication of the ACM, July, 1994.

[3] Johnston, W., Jin, G., Hoo, G., Larsen, C., Lee, J., Tierney, B.,Thompson, M. “Real-Time Digital Libraries based on Widely Distributed, High Performance Management of Large Data Objects.”, to be published in International Journal of Digital Libraries Special Issue on “Digital Libraries in Medicine”,1997. (http://www-itg.lbl.gov/WALDO/LargeDataObj-Arch.pics.fm.ps)

[4] Russell, Stuart and Norvig, Peter, Artificial Intelligence: A Modern Approach. Prentice Hall, Upper Saddle River, New Jersey, 1995.

[5] Tierney, B., Johnston, W., Chen, L.T., Herzog, H., Hoo, G., Jin, G., Lee, J., “Using High Speed Networks to Enable Distributed Parallel Image Server Systems”, Proceedings of Supercomputing ‘94, Nov. 1994, LBL-35437. (http://www-itg.lbl.gov/ISS/papers.html.)

[6] Weld, Daniel S. “An Introduction to Least Commitment Planning”, AI Magazine, Summer/Fall, 1994.